Distributed Systems Concepts and System Design

Table of Contents

[Some numbers to remember for Back of the envelope estimation 1](#_Toc139379126)

[Sequencer 1](#_Toc139379127)

[Objectives 1](#_Toc139379128)

[Requirements: 1](#_Toc139379129)

[Solutions without considering casuality: 2](#_Toc139379130)

[Solutions with casuality: 2](#_Toc139379131)

[Distributed Monitoring 3](#_Toc139379132)

[Requirements: 3](#_Toc139379133)

[Distributed Cache 4](#_Toc139379134)

[Basics: 4](#_Toc139379135)

[Writing Policies: 4](#_Toc139379136)

[Eviction Policies: 4](#_Toc139379137)

[Cache Invalidation: 4](#_Toc139379138)

[Storage Mechanism: 4](#_Toc139379139)

[Requirements: 4](#_Toc139379140)

[High Level Design: 4](#_Toc139379141)

# Some numbers to remember for Back of the envelope estimation

1. 1 MB read from memory => 9 microseconds

1 MB read from SSD => 200 microseconds

1 MB read from disk sequentially => 2 milliseconds + disk seek time = 4 milliseconds

East coast to west coast packet transfer => 71 milliseconds

1. A typical server used by Amazon => RAM : 256 GM, Storage : 15 TB, Processor: 36 cores and 72 hardware threads
2. Estimate Requests per second of a typical server: CPU bound requests and RAM bound requests. Assumption: a CPU bound request takes around 200 milliseconds to complete a request while a memory bound request takes 50 ms. A worker consumers 300 MB of ram to complete a request:
   1. So RPS for CPU bound requests = Number of hardware threads \* (1/process time) = 72 \* 1/200 = **360 RPS**.
   2. RPS for RAM bound requests = Worker Memory /​RAMsize​​ x 1/Task time = 16k RPS. So basically at any point of time how many tasks can be there based on each worker using 300 MB of RAM.
   3. So if number of CPU and RAM bond requests is same, we can have around **8k requests per second**.
3. Divide DAU / RPS to get the estimate on the number of servers.

# Sequencer

Objectives:

1. Globally unique ID for a large number of events. Uniquely identify objects like tweetId etc
2. Helpful for tracing the execution flow of events. Each event has a trace ID which can perform hundreds of microservices calls to fulfill a user request. This helps with identifying the flow of event in logs. For example, posting a video on FB. You can generate one traceId for this event which goes through a lot of microservices calls and flow can be understood by looking at the traceId in the logs.

## Requirements:

**Uniqueness**. **Scalability** – Billions of IDs per day. **Availability** – System should be working at scale i.e generate IDs for all the events that occur. Availability doesn’t concern performance. It means that if the event happens, we generate ID for it and that may be slow. If it is scalable, with large scale our performance is good so we are able to generate more IDs per day. **64 bit numeric ID –** Usually it’s good enough for billions of IDs per day.

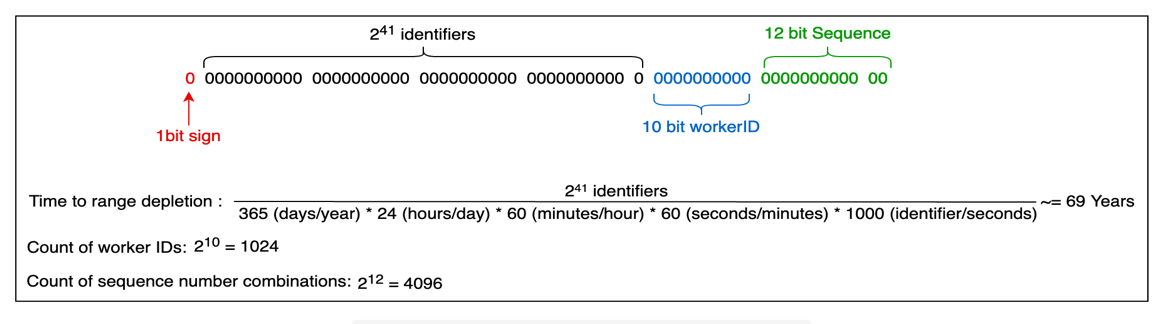
## Solutions without considering casuality:

1. **UUID (Universally) unique IDs**: 128 bit number. UUIds have different versions. Version 4 is a random number. The probability of a collision is extremely low. So essentially each node (UUID server) generates a random number and gives to a service who needs it. Pro – Doesn’t require any synchronization between the servers and is very scalable and available. Con – 128 bit makes primary key indexing slower because of massive range there are more updates on B+ tree internal nodes. So insert could be slow. Also unique cannot be guaranteed though the chances of collision are pretty low.
2. **Using a DB:** Centralised DB cannot work because of a single point of failure. Use a set of DBs each with a different start point and incrementing the counter by total number of database servers. This is scalable as we can add more servers. Cons – Adding and removing a server can generate duplicate IDs. So sometimes uniqueness condition is violated.
3. **Using a Range Handler:** Multiple Range servers which allocate unique IDs to the clients. When they start, they get a range of IDs from a centralized DB or range handler microservice. The state i.e. what server has received what range can be stored in DB and the range handler microservice can use that to allocate ranges to the range server. Pros – Scalable, Available, 64 bit and unique. Cons – We lose a significant range when a server dies.

## Solutions with casuality:

If event A happens before event B or eventB is dependent on eventA, then the ID for A < B for example tweet comments on a tweet. Physical or logical clocks are used to infer casuality.

1. **Unix timestamps** – granular to millisecond. We can have multiple servers and each server can append it’s serverId with Unix timestamp to make it scalable. Pros – simple, scalable. Cons – if two events happen in the same millisecond’th then we can have duplicates if both these concurrent events are assigned the ID by the same unix timestamp server. Scalability is still weak because we have less then billion IDs per day using this.
2. **Twitter Snowflake** -



41 bits for milliseconds. So each millisecond is a distinct identifier. For each millisecond we can have 2^10 \* 2^12 unique event IDs. Pros – Scalable, Sortable, available. Cons – IDs generated in the dead period will be wasted since no event will use them. Also they use physical clocks which are susceptible to clock drifts which can be upto 17 seconds per day. If the system uses NTP, then NTP could slow down a clock later so some ID could have been generated in future on a server. So having an accurate time is still an issue. So it has weak casuality.

1. **Logical Clocks:**
   1. **Lamport Clocks:** Each node has a counter. Event reaching node1 gets counter incremented to 1. The message sent from this event to node2 i.e. dependent events will have the counter 1 embedded in the message. When node2 receives the message, it updates the counter to max( my own, message\_counter) + 1. So we can infer partial casuality. By looking at timestamps, one cannot determine whether there is a causal relationship between two events. For example, just because event a has a timestamp of 5 and event b has a timestamp of 6, it does not imply that event a happened before event b..
   2. **Vector Clocks:** Maintain global casual history i.e. happened before relationships. Now we can just look at a timestamp and determine if they are casual or concurrent. Two events are concurrent if one vector timestamp is neither greater than nor less than the other element when doing an element-by-element comparison.Cons – Not scalable because it needs n participating nodes which can go out of hand.

Chart

Description automatically generated

1. **TrueTime API:** Only available for nodes in GCP. On calling this API, the it returns [earlies, latest]. Basically each data center has a TrueTime Server and each node syncs it’s value with the server every 30 seconds or so. Google deploys a GPS receiver or atomic clock in each data center, and clocks are synchronized within about 7 ms.Based on the drift speed of the clock on the local node, we can determine error and that is what the TrueTime API gives. 200 ppm drift typically corresponds to 7 ms error at the end of 30 seconds cycle when the node is about to sync with the master.

Graphical user interface, diagram

Description automatically generated

41 bits for earliest time, 4 bits for uncertainty, 10 for worker ID and 8 bytes for sequence number. Cons – Intervals can overlap and we cannot infer casuality then. Spanner deals with this by waiting for the delta period and using the latest timestamp as the commit timestamp for the transaction. So when transaction B starts on node B, its truetime [earliest, latest] will definitely not coincide with the range for transactionA and thus we can clearly infer that transaction A happens before transaction B. This principle is used to implement MVCC in Google Spanner and to read a lot of data without any locks. Any values for a row with the commit timestamp greater than the read time can be ignored because we have a lot of trust in the exact timestamp values now since commit waits for a certain time**. If read happens after commit, the transaction’s [earliest, latest] timestamp will be out of bounds with the [earliest, latest] timestamp for the read so these values (rows affected by the commit) can be safely read.**

**Table

Description automatically generated**

# Distributed Monitoring

## Requirements:

Monitor usage per process on a server – CPU, Memory, Disk and Network I/O, Monitor same metrics for overall server, Can server reach out of server criticsl services like NFS, monitor switches, LBs and other specialized components in a data center, latency inside and across data centers, power consumption at rack, server and data center levels.

# Distributed Cache

## Basics:

The **order** (**Writing policy**) in which data is written to cache or DB. Most frequent data in cache because of limited storage of RAM – **eviction policies**. **Cache Invalidation** once the data is outdated. Which data is stored in which server – **Storage mechanism**

## Writing Policies:

1. Write through cache – Writes in both DB and cache. Concurrently or one after other. Increases latency but highly consistent.
2. Write back cache – The data is written to cache and replicated to DB asynchronously. Low latency. But inconsistency if stale data is read from DB
3. Write around cache – The data is written to DB only. Later replicated to cache after a cache miss. Not a good strategy for reading recently updated data because of the cache miss hit. So it is not optimal for the patterns when data written is being read back as well.

## Eviction Policies:

Need to evict “cold” data i.e. less frequently accessed data. LRU – just keep the latest data in front. LFU – keep the most frequently used data in the front.

## Cache Invalidation:

How do you identify stale entries. Keep some metadata with the cache entries. TTL. Active Expiration – check the TTL actively using a daemon process. Passive Expiration – Check TTL at the time of access.

## Storage Mechanism:

Which data should be stored in which server? – Consistent Hashing. Which data structure should be used? – Simple Hash table can work sometimes for example, Ordered Dictionary for LRU cache. The value corresponding to the key points to a node in the double linked list which can later be moved to the beginning of the list. We can evict the entry at the end of the list. All the operations in O(1). Bloom filters are also used to determine if the cache entry exists in the cache server. Absence is deterministic but the presence based on bloom filters is probabilistic.

## Requirements:

1. Insert( key, value) & Retrieve(key)
2. Performance – Read and Write should be fast. Scalable – Scale horizontally with the number of requests. Affordability – Commodity Hardware. Availability – If cache is unavailable, load on DB which can then also go down. Consistency (Eventual) – Multiple cache servers may hold the same data. The cache client should ideally get the same data for the same key from them. Data sharding is done if the data to be stored in cache is huge. Number of shards depend on the amount of data and the frequency of access. Why frequency of access? If there is only one shard storing a particular popular key, then it can have high load. So it might require more replicas. Because of the replicas – the number of physical shards also increase.

## High Level Design:

A picture containing diagram, text, plan, screenshot

Description automatically generated

The cache clients reside with application servers. Configuration service ensures that all the cache clients see consistent view of cache servers. Zookeper? How the cache clients direct the requests to the cache server using the concept of consistent hashing and vnodes remain same as Cassandra. Vnodes help with uniform distribution. Multiple replicas for a particular key improves availability.